Object Detection using TensorFlow and COCO Pre-Trained Models

Introduction:

This tutorial is inspired from the research paper published by Cornell University Library, in this we are going to explore how to use TensorFlow's Object Detection API to train your own convolutional neural network object detection classifier for multiple objects on Windows 10, 8, or 7, starting from scratch. At the end of this we will have a program that can identify and draw boxes around specific objects in pictures, videos, or in a webcam feed.

Understanding Architecture:

In this tutorial we used Faster R-CNN Model, so let's download & understand in-depth about the Faster-RCNN-Inception-V2 model architecture, how it works and visualize the output by training on our own dataset.

Now what is Faster-RCNN?

It's a network that does object detection. As explained by its name it's faster than its descendants RCNN and FastRCNN. This network has use cases in self-driving cars, manufacturing, security, and is even used at Pinterest.

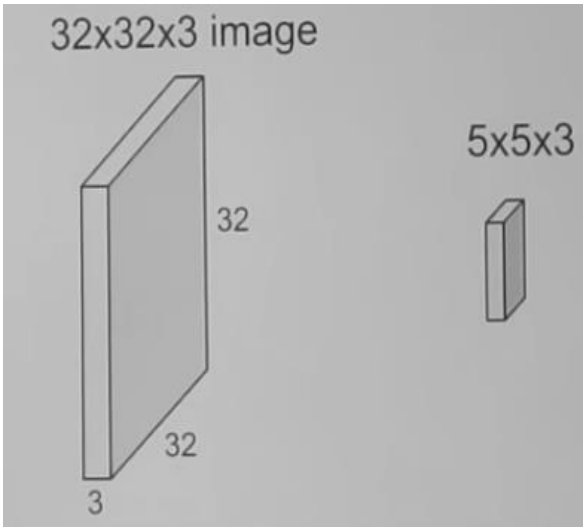
How it works?

1) Run the image through a CNN to get a Feature Map
2) Run the Activation Map through a separate network, called the Region Proposal Network (RPN), that outputs interesting boxes/regions
3) For the interesting boxes/regions from RPN use several fully connected layer to output class + Bounding Box coordinates

So to move further in Faster-RCNN we should be clear in our understanding on CNN, let's start our exploration of CNN.

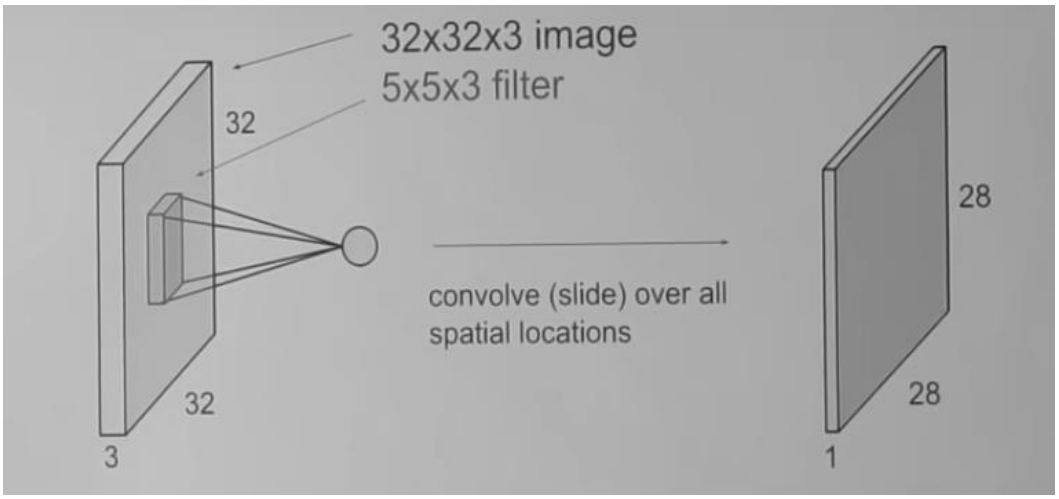
Working of CNN:

Unlike neural networks, the input is a vector with a multi-channelled image (3 channelled- RGB in this case).



What is Convolution?

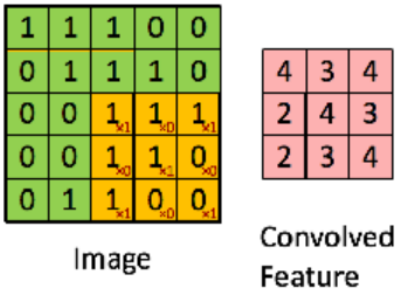
Convolution is the first layer to extract features from an input image. It preserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs such as image matrix and a filter or kernel.



From the above image we can observe that for our input of 32*32*3 we took a filter of 5*5*3 and slided it over the complete image and along the way take the dot product between the filter and chunks of the input image. The output results with an image of size 28*28*1.

E.g.: Consider a 5 x 5 whose image pixel values are 0, 1 and filter matrix 3 x 3 as shown in below

Then the convolution of 5 x 5 image matrix multiplies with 3 x 3 filter matrix which is called “Feature Map” as output shown in below with a stride of 1.



Stride: Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on.

Padding: If we increase the stride value the size of image keeps on reducing, padding with zeros across it solves this problem.

From the below image we can observe the size of image is retained after padding zeros with stride 1.

0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139	184	250	409	410	283
133	429	505	686	856	441
310	261	792	412	640	341
280	633	653	851	751	317
254	608	913	713	657	503
321	325	592	517	637	78

ReLU Activation: ReLU stands for Rectified Linear Unit for a non-linear operation. Its purpose is to introduce non-linearity which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function in our ConvNet. Since, the real world data would want our ConvNet to learn would be non-negative linear values.

Pooling: Sometimes when the images are too large, we would need to reduce the number of trainable parameters. It is then desired to periodically introduce pooling layers between subsequent convolution layers. Pooling is done for the sole purpose of reducing the spatial size of the image. Pooling is done independently on each depth dimension, therefore the depth of the image remains unchanged. The most common form of pooling layer generally applied is the max pooling.

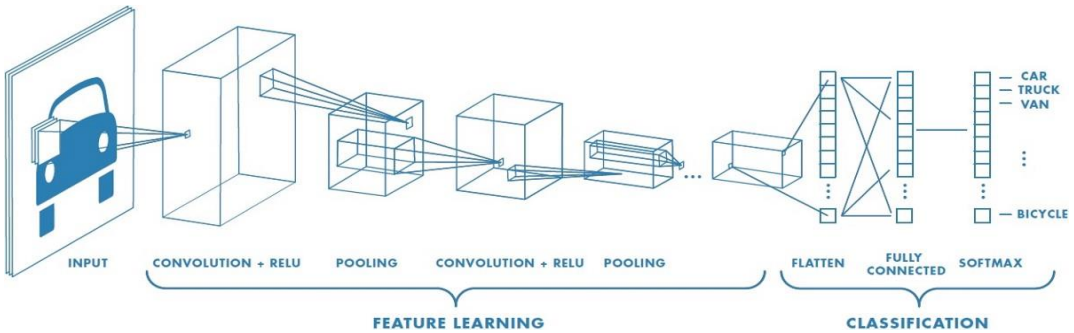
E.g.:

429	505	686	856
261	792	412	640
633	653	851	751
608	913	713	657

792	856
913	851

Here we have taken stride as 2, while pooling size also as 2. The max operation is applied to each depth dimension of the convolved output. As you can see, the 4*4 convolved output has become 2*2 after the max pooling operation.

The below figure is a complete flow of CNN to process an input image and classifies the objects based on values.

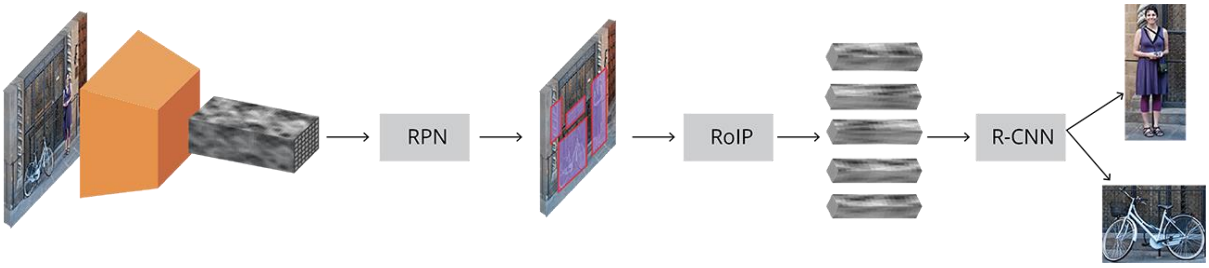


Fully Connected Layer/Output Layer:

The feature map matrix after pooling layer will be flattened as vector (x1, x2, x3 …). With the fully connected layers, we combined these features together to create a model. Convolution layers generate 3D activation maps while we just need the output as whether or not an image belongs to a particular class. The output layer has a loss function like categorical cross-entropy, to compute the error in prediction. Once the forward pass is complete the backpropagation begins to update the weight and biases for error and loss reduction.

Now after getting a feature map from the CNN, we need to pass it to Region Proposal Network (RPN), to find interesting regions. It returns regions of interest, the loss regarding whether it found an object and the loss regarding the location of the object.

Following is the high level architecture for Faster-RCNN



Once we got the features for the input image from CNN, generation of region proposals (Anchors/Bounding Box) can be done with Region Proposal Network (RPN) layer. The predicted region proposals are then reshaped using a Region of Interest Pooling (RoIP) layer which is then used to classify the image within the proposed region and predict the offset values with R-CNN for the bounding boxes.

R-CNN: It's a method where we use selective search to extract just 2000 regions from the image (region proposals). Therefore, now, instead of trying to classify a huge number of regions, you can just work with 2000 regions. These 2000 candidate region proposals are warped into a square and fed into a CNN that produces a 4096-dimensional feature vector as output. The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into an SVM to classify the presence of the object within that candidate region proposal. In addition to predicting the presence of an object within the region proposals, the algorithm also predicts four values which are offset values to increase the precision of the bounding box. For example, given a region proposal, the algorithm would have predicted the presence of a person but the face of that person within that region proposal could've been cut in half. Therefore, the offset values help in adjusting the bounding box of the region proposal.

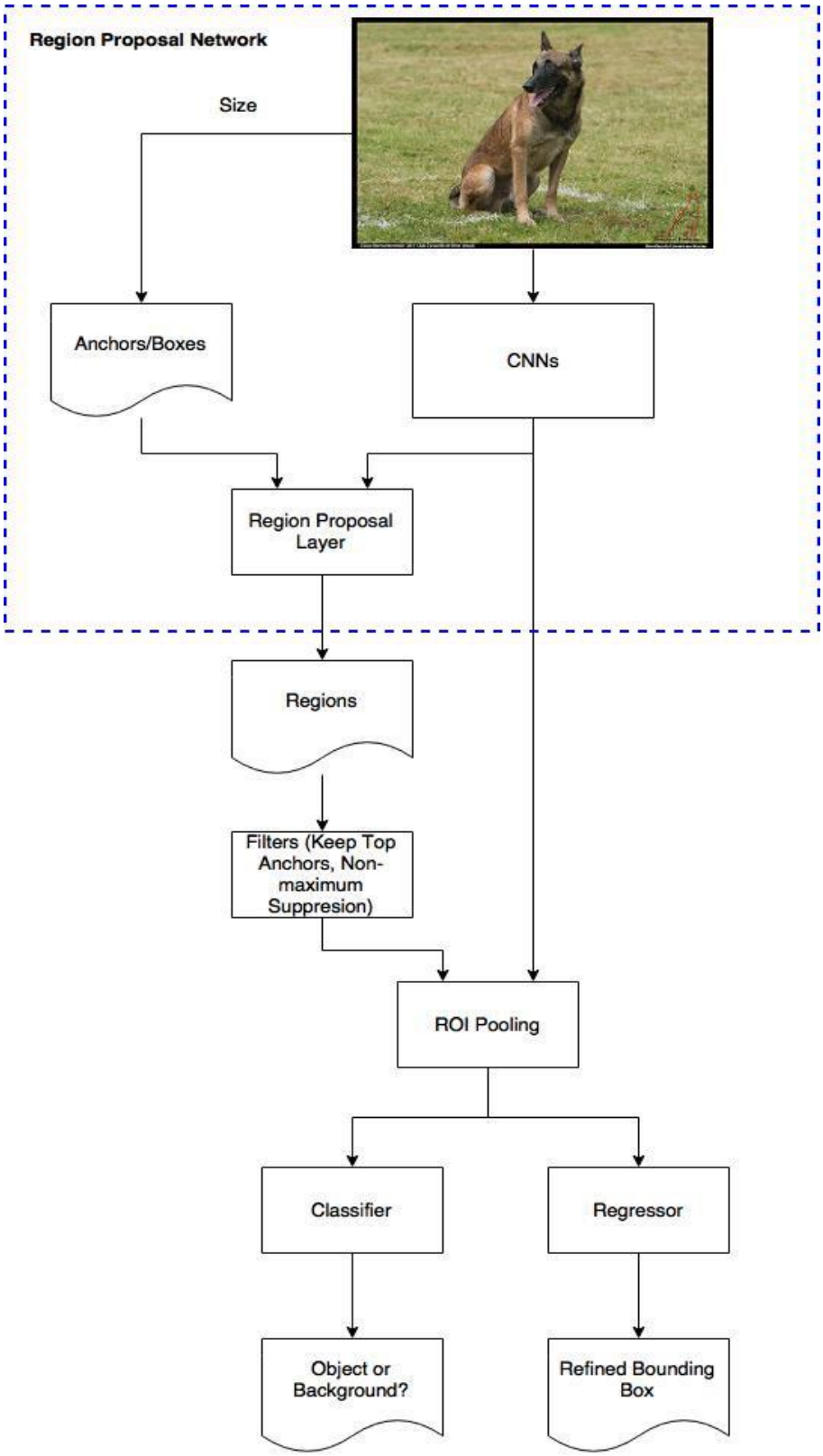
Fast R-CNN: Instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map. From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a RoIP layer we reshape them into a fixed size so that it can be fed into a fully connected layer. From the RoI feature vector, we use a softmax layer to predict the class of the proposed region and also the offset values for the bounding box.

The reason "Fast R-CNN" is faster than R-CNN is because you don't have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it.

Why it's faster than its predecessors?

Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals.

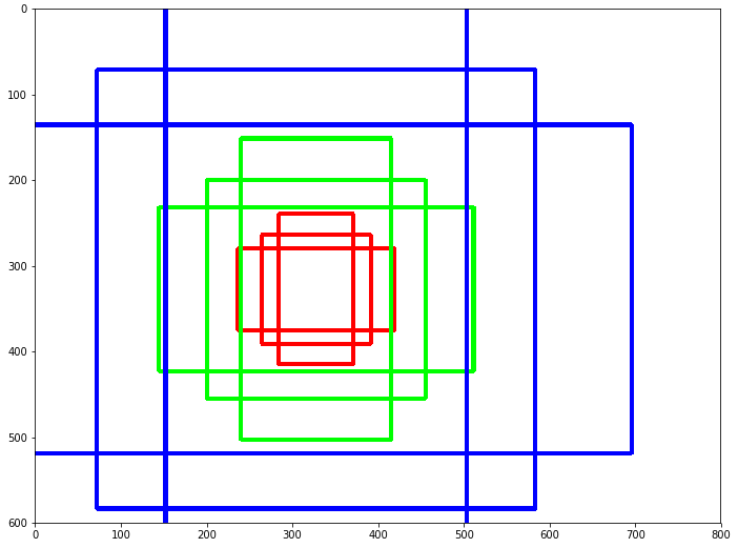
Architecture of Faster R-CNN:



The above architecture will explain the sequence of operations performed once an image got its feature matrix from CNN layer.

Working of RPN:

After getting a set of convolutional feature maps (feature matrix) on the last convolutional layer a sliding window is run spatially on these feature maps. The size of sliding window is n×n (here 3×3). For each sliding window, a set of 9 anchors are generated which all have the same center (xa, ya) but with 3 different aspect ratios and 3 different scales as shown below.



Let's look closer:

1. Three colours represent three scales or sizes: 128x128, 256x256 and 512x512.
2. Let's single out the red boxes/anchors. The three boxes have height width ratios 1:1, 1:2 and 2:1 respectively.

These anchors work well for Pascal VOC dataset as well as the COCO dataset. However, you have the freedom to design different kinds of anchors/boxes. For example, you are designing a network to count passengers/pedestrians, you may not need to consider the very short, very big, or square boxes. A neat set of anchors may increase the speed as well as the accuracy.

Furthermore, for each of these anchors, a value $p*$ is computed which indicated how much these anchors overlap with the ground-truth bounding boxes (GTBox).

$$\text{IoU} = \frac{\text{A n Gt}}{\text{A u Gt}} \begin{cases} > 0.7 = \text{object} \\ < 0.3 = \text{not object} \end{cases}$$

where ***IoU*** is intersection over union and is defined below:

$$IoU = \frac{Anchor \cap GTBox}{Anchor \cup GTBox}$$

Finally, the 3×3 spatial features extracted from those convolution feature maps are fed to a smaller network which has two tasks: classification and regression. The output of regressor determines a predicted bounding-box (x,y,w,h), the output of classification sub-network is a probability p indicating whether the predicted box contains an object (1) or it is from background (0 for no object).

Working of ROI Pooling Layer:

After RPN, we get proposed regions with different sizes. Different sized regions mean different sized CNN feature maps. It's not easy to make an efficient structure to work on features with different sizes. ROI Pooling can simplify the problem by reducing the feature maps into the same size.

The result is that from a list of rectangles with different sizes we can quickly get a list of corresponding feature maps with a fixed size.

Benefit of RoI pooling? One of them is processing speed. If there are multiple object proposals on the frame (and usually there'll be a lot of them), we can still use the same input feature map for all of them. Since computing the convolutions at early stages of processing is very expensive, this approach can save us a lot of time.

Final layer R-CNN:

Region-based convolutional neural network (R-CNN) is the final step in Faster R-CNN's pipeline. After getting a convolutional feature map from the image, using it to get object proposals with the RPN and finally extracting features for each of those proposals (via RoI Pooling), we finally need to use these features for classification. R-CNN tries to mimic the final stages of classification CNNs where a fully-connected layer is used to output a score for each possible object class.

R-CNN has two different goals:

1. Classify proposals into one of the classes, plus a background class (for removing bad proposals).
2. Better adjust the bounding box for the proposal according to the predicted class.

Implementation:

Now let's start implementing by taking a use case with some sample dataset.

Use case: Train some sample images of Shirts, T-Shirts and Jeans, so that out model should recognize the Shirt, T-Shirt and Jeans from the given image.

Dataset Preparation: We downloaded some sample images from google and assigned labels for those images. Labelling can be done with the tool called labelimg.

Things to setup before start modelling:

Installation:

Step 1: Download and install Anaconda with Python 3.6/3.7 here and create a virtual environment by issuing the following commands respectively.

1. C:\> conda create -n tensorflow1 pip python=3.5
2. C:\> activate tensorflow1

Step 2:

1. System with NVIDIA Graphics: Install TensorFlow-GPU, CUDA and cuDNN by following the given steps.
2. System with normal Intel Graphics: Install regular TensorFlow and you do not need to install CUDA and cuDNN.

TensorFlow installation can be done by using “(tensorflow1) C:\> pip install --ignore-installed --upgrade tensorflow-gpu” for GPU versions and “(tensorflow1) C:\> pip install --ignore-installed --upgrade tensorflow” for regular versions.

Note*: If you are installing TensorFlow-GPU v1.4 make sure to install CUDA v9.0 and cuDNN v7.0 rather than CUDA v8.0 and cuDNN v6.0 because they are supported by TensorFlow-GPU v1.5.

Step 3: Install the other necessary packages by issuing the following commands:

(tensorflow1) C:\> conda install -c anaconda protobuf

(tensorflow1) C:\> pip install pillow

(tensorflow1) C:\> pip install lxml

(tensorflow1) C:\> pip install Cython

(tensorflow1) C:\> pip install jupyter

(tensorflow1) C:\> pip install matplotlib

(tensorflow1) C:\> pip install pandas

(tensorflow1) C:\> pip install opencv-python

Before moving forward let’s understand the libraries briefly.

Protocol Buffers (protobuf): It’s a method of translating image files into a format that can be stored in a file, useful in developing programs to communicate with each other over a wire or for storing data. It will emphasize simplicity and performance.

Pillow: Python Imaging Library (abbreviated as PIL) (in newer versions known as Pillow) is a free library for the Python programming language that adds support for opening, manipulating, and saving many different image file formats.

LXML: Helps in processing XML and HTML in the Python language.

Cython: Gives C-like performance with code that is written mostly in Python with optional additional C-inspired syntax.

Jupiter: Interactive python web application used to create, share, view and compile code.

Matplot: Plotting library for python helps in providing plots.

Pandas: Python Data Analysis Library.

Open-CV: It’s a library of Python bindings designed to solve computer vision problems.

Step 4: Download tensorflow Object Detection API repository from GitHub

Create a working directly in C: and name it “tensorflow1”, it will contain the full TensorFlow object detection framework, as well as your training images, training data, trained classifier, configuration files, and everything else needed for the object detection classifier.

Download the full TensorFlow object detection repository here, open the downloaded zip file and extract the “models-master” folder directly into the C:\tensorflow1 directory you just created. Rename “models-master” to just “models”.

Step 5: Download a pre-trained object detection models on COCO dataset, the Kitti dataset, the Open Images dataset, the AVA v2.1 dataset and the iNaturalist Species Detection Dataset from the tensorflow model zoo git reposit.

After landing into the model zoo git repo we can see multiple models start from ssd_mobilenet_v1_coco to mask_rcnn_resnet50_atrous_coco.

The architecture of each model in this repo is unique which allows for faster/slower detections and more/less accuracy etc.

Some models (such as the SSD-MobileNet model) have an architecture that allows for faster detection but with less accuracy, while some models (such as the Faster-RCNN model) give slower detection but with more accuracy.

In the table of models, we can observe a column called **COCO mAP[^1]** means **Mean Average Precision**. It's a metric to measure the accuracy of object detectors. It's an average of the maximum precisions at different recall values. To understand more click here.

Once the desired model is downloaded extract its contents to C:\tensorflow1\models\research\object_detection

Download the full repository located on this page (scroll to the top and click Clone or Download) and extract all the contents directly into the C:\tensorflow1\models\research\object_detection directory. (You can overwrite the existing "README.md" file.)

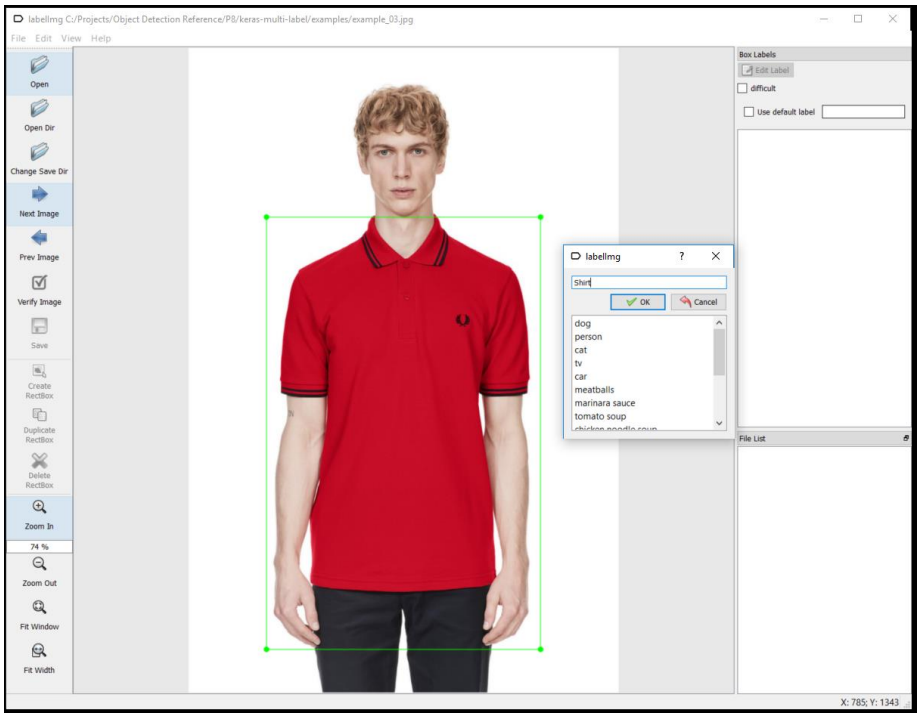
Step 6: Clean-up

Go to folder named research and then remove all the folders except ***Object Detection, Slim*** folders and ***set.py*** file**.**

To train your own object detector, delete the following files (do not delete the folders):

1) All files in \object_detection\images\train and \object_detection\images\test
2) The “test_labels.csv” and “train_labels.csv” files in \object_detection\images
3) All files in \object_detection\training
4) All files in \object_detection\inference_graph

Paste our dataset in train and test folders respectively and label them using Labelimg tool as shown below.



An xml file contains the label data will be generated for each image and stored in train and test folders respectively.

A PYTHONPATH variable must be created that points to the \models, \models\research, and \models\research\slim directories. Do this by issuing the following commands (from any directory):

1) **(tensorflow1) C:\>set PYTHONPATH=C:\tensorflow1\models;C:\tensorflow1\models\research;C:\tensorflow1\models\research\slim**
2) **(tensorflow1) C:\>set PATH=%PATH%;PYTHONPATH**
3) **(tensorflow1) C:\>echo %PATH%**
4) **(tensorflow1) C:\>echo %PYTHONPATH%**

Compile the Protobuf files, which are used by TensorFlow to configure model and training parameters. Every .proto file in the \object_detection\protos directory must be called out

individually by the command. Before you hit the command you should be in this path \research and issue the following command:

"(tensorflow1) C:\>tensorflow1\models\research>protoc --python_out=.
.\object_detection\protos\anchor_generator.proto
.\object_detection\protos\argmax_matcher.proto
.\object_detection\protos\bipartite_matcher.proto .\object_detection\protos\box_coder.proto
.\object_detection\protos\box_predictor.proto .\object_detection\protos\eval.proto
.\object_detection\protos\faster_rcnn.proto
.\object_detection\protos\faster_rcnn_box_coder.proto
.\object_detection\protos\grid_anchor_generator.proto
.\object_detection\protos\hyperparams.proto .\object_detection\protos\image_resizer.proto
.\object_detection\protos\input_reader.proto .\object_detection\protos\losses.proto
.\object_detection\protos\matcher.proto
.\object_detection\protos\mean_stddev_box_coder.proto
.\object_detection\protos\model.proto .\object_detection\protos\optimizer.proto
.\object_detection\protos\pipeline.proto .\object_detection\protos\post_processing.proto
.\object_detection\protos\preprocessor.proto
.\object_detection\protos\region_similarity_calculator.proto
.\object_detection\protos\square_box_coder.proto .\object_detection\protos\ssd.proto
.\object_detection\protos\ssd_anchor_generator.proto
.\object_detection\protos\string_int_label_map.proto .\object_detection\protos\train.proto
.\object_detection\protos\keypoint_box_coder.proto
.\object_detection\protos\multiscale_anchor_generator.proto
.\object_detection\protos\graph_rewriter.proto"

This creates a name_pb2.py file from every name.proto file in the \object_detection\protos folder.

(Note: TensorFlow occasionally adds new .proto files to the \protos folder. If you get an error saying ImportError: cannot import name 'something_something_pb2', you may need to update the protoc command to include the new .proto files.)

Step 7: Run the following commands from C:\tensorflow1\models\research directory

1. **(tensorflow1) C:\tensorflow1\models\research> python setup.py build**
2. **(tensorflow1) C:\tensorflow1\models\research> python setup.py install**

Step 8: Generate Training and Test Data

Convert all the .xml files in both training and test folders into .csv files by using below command.

(tensorflow1) C:\tensorflow1\models\research\object_detection> python xml_to_csv.py

This creates a train_labels.csv and test_labels.csv file in the models\research\object_detection\images directory.

Step 9: Generate .tf record

Open the generate_tfrecord.py file from object_detection folder in a text editor. Replace the label map starting at line 31 with your own label map, where each object is assigned an ID number.

```
def class_text_to_int(row_label):
    if row_label == 'Shirt':
        return 1
    elif row_label == 'T-Shirt':
        return 2
    elif row_label == 'Jeans':
        return 3
    else:
        return None
```

Then, generate the TFRecord files by issuing these commands from the \object_detection folder:

1. **python generate_tfrecord.py --csv_input=images\train_labels.csv --image_dir=images\train --output_path=train.record**
2. **python generate_tfrecord.py --csv_input=images\test_labels.csv --image_dir=images\test --output_path=test.record**

Step 10: Create Label Map and Configure Training

The label map tells the trainer what each object is by defining a mapping of class names to class ID numbers. Use a text editor to create a new file and save it as ***labelmap.pbtxt***
in C:\tensorflow1\models\research\object_detection\training folder the file type should be .pbtxt

The label map ID numbers should be the same as what is defined in the generate_tfrecord.py file, the labelmap.pbtxt file will look like below:

```
item {
  id: 1
  name: 'Shirt'
}

item {
  id: 2
  name: 'T-Shirt'
}

item {
  id: 3
  name: 'Jeans'
}
```

Step 11: Configure Training

The object detection training pipeline should be configured. It defines which model and what parameters will be used for training. Navigate to C:\tensorflow1\models\research\object_detection\samples\configs and copy the faster_rcnn_inception_v2_pets.config file into the \research \object_detection\training directory.

Open faster_rcnn_inception_v2_pets.config file in a text editor. Make some necessary changes to the .config file, mainly changing the number of classes and examples, and adding the file paths to the training data.

Make the following changes to the faster_rcnn_inception_v2_pets.config file.

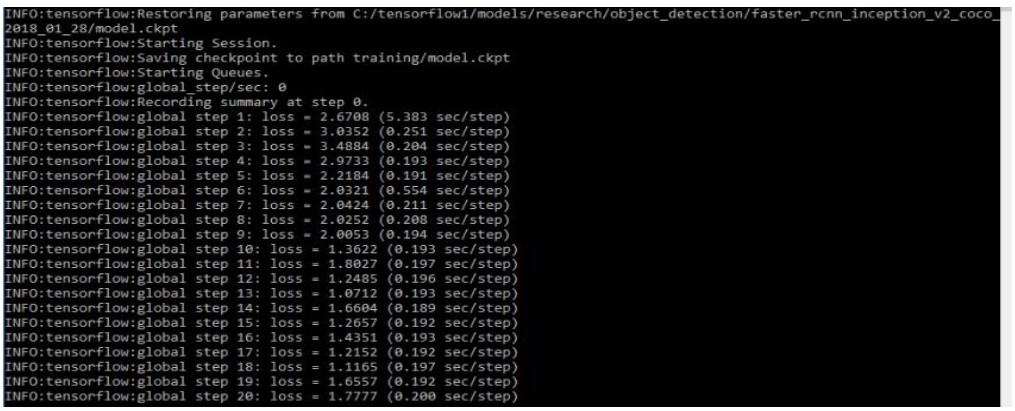
- Line 9. Change num_classes to the number of different objects you want the classifier to detect. For the above shirt, t-shirt and jeans detector, it would be num_classes: 3.
- Line 106. Change fine_tune_checkpoint to:
    - **fine_tune_checkpoint: "C:/tensorflow1/models/research/object_detection/**
    - **faster_rcnn_inception_v2_coco_2018_01_28/model.ckpt"**
- Lines 123 and 125. In the train_input_reader section, change input_path and label_map_path to:
    - **input_path: "C:/tensorflow1/models/research/object_detection/train.record"**
    - **label_map_path: "C:/tensorflow1/models/research/object_detection/training/labelmap.pbtxt"**
- Line 130. Change num_examples to the number of images you have in the \images\test directory.
- Lines 135 and 137. In the eval_input_reader section, change input_path and label_map_path to:
    - **input_path: "C:/tensorflow1/models/research/object_detection/test.record"**
    - **label_map_path: "C:/tensorflow1/models/research/object_detection/training/labelmap.pbtxt"**

Save the file after the changes have been made. The training job is all configured and ready to run.

Model Training: In the \object_detection path put the following command to begin the training.

python train.py --logtostderr --train_dir=training/ --pipeline_config_path=training/faster_rcnn_inception_v2_pets.config

TensorFlow will initialize the training. When training begins, it will look like this:



Each step of training reports the loss, eventually it will come down as training progresses. It's recommend to allow your model to train until the loss consistently drops below 0.05.

You can view the progress of the training job by using TensorBoard. To do this, open a new instance of Anaconda Prompt, change to the C:\tensorflow1\models\research\object_detection directory, and issue the following command:

(tensorflow1) C:\tensorflow1\models\research\object_detection>tensorboard --logdir=training

This will create a webpage on your local machine at YourPCName:6006, which can be viewed through a web browser. The TensorBoard page provides information and graphs that show how the training is progressing. One important graph is the Loss graph, which shows the overall loss of the classifier over time.

The training routine periodically saves checkpoints about every five minutes. You can terminate the training by pressing Ctrl+C while in the command prompt window. It's recommended to wait until just after a checkpoint has been saved to terminate the training. You can terminate training and start it later, and it will restart from the last saved checkpoint. The checkpoint at the highest number of steps will be used to generate the frozen inference graph.

Export Inference Graph: After training is complete, the last step is to generate the frozen inference graph (.pb file). From the \object_detection folder, issue the following command, where "????" in "model.ckpt-????.meta" should be replaced with the highest-numbered .ckpt file in the training folder:

python export_inference_graph.py --input_type image_tensor --pipeline_config_path training/faster_rcnn_inception_v2_pets.config --trained_checkpoint_prefix training/model.ckpt-???? --output_directory inference_graph

This creates a frozen_inference_graph.pb file in the \object_detection\inference_graph folder. The .pb file contains the object detection classifier.

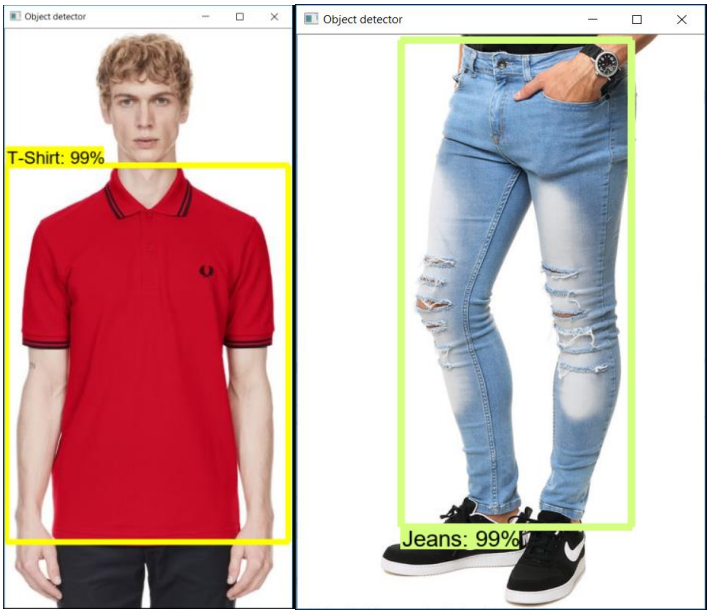
About the type of Modelling: The entire process of using a pre-trained model created by someone else to solve a similar problem, this type of learning is called *Transfer Learning*.

Advantage: A pre-trained model may not be 100% accurate in your application, but it saves huge efforts required to re-invent the wheel.

Output:

To test your object detector, move a picture of the object or objects into the \object_detection folder, and change the IMAGE_NAME variable in the Object_detection_image.py to match the file name of the picture. Alternatively, you can use a video of the objects (using Object_detection_video.py), or just plug in a USB webcam and point it at the objects (using Object_detection_webcam.py).

If everything is working properly, the object detector will initialize for about 10 seconds and then display a window showing any objects it's detected in the image!



PS: I hope you got an idea on the architecture and implementation.

Authors
GopiKrishna
Gnaneswarrao
Sowjanya